Cryptographic Verification by Typing for a Sample Protocol Implementation

Cédric Fournet, Karthikeyan Bhargavan, Andrew D. Gordon

Cryptographic Verification by Typing for a Sample Protocol Implementation

Cédric Fournet[1], Karthikeyan Bhargavan[2], and Andrew D. Gordon[1,3]

[1] Microsoft Research
[2] INRIA Rocquencourt
[3] University of Edinburgh

Abstract. Type systems are effective tools for verifying the security of cryptographic protocols and implementations. They provide automation, modularity and scalability, and have been applied to large protocols. In this tutorial, we illustrate the use of types for verifying authenticity properties, first using a symbolic model of cryptography, then relying on a concrete computational assumption.

(1) We introduce refinement types (that is, types carrying formulas to record invariants) for programs written in F# and verified by F7, an SMT-based type checker.
(2) We describe a sample authenticated RPC protocol, we implement it in F#, and we specify its security against active adversaries.
(3) We develop a sample symbolic library, we present its main cryptographic invariants, and we show that our RPC implementation is perfectly secure when linked to this symbolic library.
(4) We implement the same library using concrete cryptographic primitives, we make a standard computational assumption, and we show that our RPC implementation is also secure with overwhelming probability when linked to this concrete library.

1 Verifying Protocol Implementations

Cryptographic Protocols Go Wrong. Security flaws still regularly appear in widely-used protocol implementations, sometimes years after their deployment, despite the best efforts of skilled designers, developers, testers, and reviewers. We may organize these flaws into three categories: logical, cryptographic, and implementation flaws.

- As a classic example of a logical flaw, consider the vulnerability in the public-key protocol of Needham and Schroeder (1978), discovered by Lowe (1996) in his seminal paper on model-checking security protocols. This man-in-the-middle attack is the staple example for protocol verification; it is well known in the formal methods research community, and many tools can discover it. Still, for instance, Cervesato et al. (2008) recently discovered that the IETF issued a public-key variant of Kerberos, shipped by multiple vendors, with essentially the same design flaw.
- As an example of a cryptographic flaw, consider the padding-oracle attacks on unauthenticated ciphertexts, discovered by Bleichenbacher (1998) and Vaudenay

Lecture notes to appear in the proceedings of the Foundations of Security and Design VI summer school (FOSAD 2010), Sept. 6–11 2010, http://www.sti.uniurb.it/events/fosad10/ A. Aldono and R. Gorrieri, ed. LNCS, 2011.

(2002). This is a well-known, practical class of side-channel attacks. Still, it involves subtle interferences between cryptographic algorithms and error handling, and remains difficult to prevent without inspecting many implementation details. For instance, Albrecht et al. (2009) recently showed that SSH implementations were still vulnerable to a similar attack.

– Less interestingly, perhaps, implementation flaws due to coding errors are still commonplace. For instance, several security updates patched flaws in the processing of X.509 certificates in TLS clients, leading to secure connections with the wrong server. In contrast with functional testing, security testing does not provide much coverage against active attacks. The difficulty lies in the details of a large amount of code that might compromise the security of a protocol, for example in the parsing of untrusted data or the handling of runtime errors.

We would like to reliably design, develop, and deploy protocols and similar programs that rely on cryptography. To this end, we build formal methods and automated tools to relate protocol implementations and cryptographic models. Before presenting one such method, based on refinement types, we briefly discuss cryptographic models and their practical application to production code.

Symbolic and Computational Models for Cryptography In theory, the main security properties of protocols are now well understood. Two main verification approaches have been successfully applied:

(1) Symbolic, "black box" models of cryptography, first proposed by Needham and Schroeder (1978) and formalized by Dolev and Yao (1983), focus on the logical structure of the protocol, and assume that their cryptographic primitives are perfect. Cryptographic values range over abstract terms, in an algebra subject to a few rewrite rules. Adversaries range over arbitrary programs that control the network and can intercept, rewrite, and inject messages. These models naturally benefit from formalisms developed for programming languages; their security can be verified using automated tools that scale up to large systems (Abadi and Blanchet 2005).

(2) Computational models of cryptography focus on algorithms and the complexity of attacks. Cryptographic values range over concrete bitstrings of bounded length. Adversaries range over probabilistic, polynomial machines. These models are more precise, but also more difficult to apply to large systems. Automated verification is more challenging, although formal tools are starting to appear.

Reconciling these two approaches is an active subject of research. Under suitable restrictions and computational assumptions, one can prove that some symbolic models are computationally sound for all protocols, then conduct program verification using off-the shelf symbolic tools. On the other hand, when considering the details of concrete protocols, the existence of a sound symbolic model is far from granted, and it is often more effective to adapt verification tools to deal directly with the cryptographic patterns and assumptions of the protocol being considered.

Specifications, Models, and Implementations In practice, production code and cryptographic models evolve independently, and interact only through informal specifications.

Specifications and standards are large text documents: their main purpose is to define message formats and promote interoperability between implementations, rather than stating security goals and detailing their local enforcement.

Models are short, abstract descriptions handwritten by specialists, using either formal specification languages or pseudo-code. They state what must be verified, but necessarily ignore the details of full-fledged implementations.

Implementations, typically coded in C or C++, deal with low-level details. Sadly, their main security concerns still are to avoid buffer overruns and maintain memory safety. Cryptographic issues are manually reviewed from time to time, but this kind of code review can hardly be applied each time the code is modified.

We propose to bridge the gap between models and implementations from both directions: we build tools that automatically verify cryptographic properties on source code, rather than simplified models; and we develop models as executable code (in this tutorial, written in F#); this code may serve as a reference implementation, aiming at simplicity rather than performance. It may additionally be used for testing other legacy implementations.

Contents This tutorial illustrates the verification of authentication properties on a sample RPC protocol coded in F#. The protocol relies on cryptographic keyed hashes, also known as MACs. It is a variant of protocols formally studied by Bhargavan et al. (2010) and Fournet et al. (2011). It is small enough to explain its code and its verification in details. We show how to verify it by typechecking against F7 interfaces, first using a symbolic model of MACs, then using a more concrete, computational assumption, namely resistance to existential forgery chosen-message attacks.

The rest of the paper is organized as follows. Section 2 reviews F7, our main verification tool. Section 3 describes our sample protocol, presents its F# implementation, and discusses its security. Section 4 develops its modular, logical specification. Section 5 shows how to verify our code against active adversaries within a sample symbolic model of cryptography. Section 6 shows how to verify the same code against active, probabilistic polynomial adversaries, within a simple computational model of cryptography. Finally, Section 7 offers points for further reading.

This tutorial omits most formal definitions. Another, complementary tutorial by Gordon and Fournet (2010) presents the refinement type system at the core of F7, and outlines its applications, but does not deal with cryptography.

A website `http://research.microsoft.com/f7` hosts additional materials, including related research papers, the latest release of the F7 tool, and the code presented in this tutorial.

2 F7: Automated Program Verification using Refinement Types

We now describe a syntax for F# programs annotated with logical specifications and refinement types and show how such programs can be verified using the F7 typechecker. This section is largely independent of the example developed in the rest of the paper. Here, we focus on the syntax and command-lines that are commonly used when verifying cryptographic protocols.

2.1 Security Programming in F#

We first give the core syntax of the fragment of F# we use for writing programs. This fragment is a standard, ML-like call-by-value functional language that includes recursive functions and recursive algebraic datatypes, plus two constructs for writing specifications. We can define data structures such as lists, options, trees, and XML documents in this language. Notably, however, our fragment does not support F# classes, objects, or looping constructs. Still, the language described here has proved adequate to program several thousand lines of security applications and large cryptographic protocols (Bhargavan et al. 2009b, 2010). We believe that for the security programmer, writing in this restricted fragment is a reasonable trade-off in exchange for automated verification support using F7.

A program is a sequence of *modules* whose syntax can be defined as follows:

Syntax of F# Modules (patterns, expressions, definitions)

$P, Q ::=$	Patterns
$(), \mathbf{true}, \mathbf{false}, i, \text{“}s\text{”}$	constants: unit, booleans, integers, strings
x	variable
$(P_1, \ldots, P_n)$	tuple
$h\ P$	constructor
$e ::=$	Expressions
$(), \mathbf{true}, \mathbf{false}, i, \text{“}s\text{”}$	constants: unit, booleans, integers, strings
x	variable
$(e_1, \ldots, e_n)$	tuple
$\mathbf{fun}\, x \rightarrow e$	function (scope of x is e)
$h\ e$	constructor application
$e_1\, e_2$	function application
$\mathbf{let}\ P = e_1\ \mathbf{in}\ e_2$	sequential composition (scope of variables in P is e_2)
$\mathbf{match}\ e\ \mathbf{with}\ \prod_i (P_i\ \rightarrow e_i)$	pattern matching (scope of variables in P_i is e_i)
$\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)$	conditional
$\mathbf{assume}\ (p\ M)$	assume that $p\ M$ is true
$\mathbf{assert}\ (p\ M)$	require that $p\ M$ must be true
$\delta ::=$	Definitions
$\mathbf{let}\ x = e$	value definition
$\mathbf{let\ rec}\ f = \mathbf{fun}\, x \rightarrow e$	recursive function definition (scope of f and x is e)
$\mathbf{type}\ \alpha = T$	type abbreviation
$\mathbf{type}\ \alpha = \prod_i (h_i : T_i)$	recursive algebraic datatype
$\mathbf{open}\ M$	import another module
$S ::= \mathbf{module}\ M\ \delta_1 \ldots \delta_n$	Module

We first state some syntactic conventions. We write $\prod_i \phi_i$ as a shorthand for a sequence $\phi_1 \mid \cdots \mid \phi_n$ of phrases separated by the $\mid$ symbol. Our phrases of syntax may contain two kinds of identifier. Type constructors, written α, refer to datatypes and type abbreviations, such as key or bytes. Program variables, written x, y, z refer to values and functions; by convention, we use f to refer to function names, h to refer to constructors, p to refer to predicate names, and l to refer to logical function names (explained later). Each variable has a scope and we identify phrases of syntax up to consistent renaming of bound variables. We write $\psi\{\phi/\iota\}$ for the capture-avoiding substitution of the phrase ϕ for each free occurrence of identifier ι in the phrase ψ.

Each module (*M*) is written in a separate file (`m.fs`) and consists of a sequence of definitions. A definition (δ) may introduce a new global variable (*M.x*, *M.f*) that represents a value or a (possibly recursive) function, or it may define a new type constructor symbol ($M.\alpha$) that represents a type abbreviation or a (possibly recursive) algebraic datatype with a set of term constructors ($M.h_i$ that take arguments of type T_i). The types that appear in type definitions and abbreviations are in standard F# type syntax (not the extended F7 syntax defined later in this section). Values and type constructors in another module (*N*) can be referred to either by using their fully-qualified names (*N.x*, *N.f*, $N.\alpha$) or by opening the module (**open** *N*).

As an illustration of the syntax, we display the module *Msg* from the file `msg.fs`

Serializing messages as byte arrays: (`msg.fs`)

```
1   module Msg
2
3   type msg =
4     | Request of string
5     | Response of string
6
7   let tag0 = Lib.utf8 "0"
8   let tag1 = Lib.utf8 "1"
9
10  let msg_to_bytes = fun m →
11    match m with
12      Request s →Lib.concat tag0 (Lib.utf8 s)
13    | Response s →Lib.concat tag1 (Lib.utf8 s)
14
15  let bytes_to_msg = fun b →
16    let (tag,sb) = Lib.split 2 b in
17    let s = Lib.iutf8 sb in
18      if tag = tag0 then Request s
19      else if tag = tag1 then Response s
20           else failwith "incorrect tag"
```

The module *Msg* defines an algebraic datatype *msg*, two values *tag0* and *tag1* and two functions *msg_to_bytes* and *bytes_to_msg*. The type *msg* has two constructors *Request* and *Response*; each takes a string as argument and constructs a value of type *msg*. The right-hand-side of each value and function definition is an expression (*e*). The expressions in *Msg* use auxiliary functions from the module *Lib*; *utf8* and *iutf8* convert between strings and byte arrays; *concat* and *split* concatenate and split byte arrays. In addition, they use two predefined functions: = (structural equality) and *failwith* (to raise an error and terminate the program).

An expression (*e*) may be a constant (2, "0"), a local or global variable (*m*, *Msg.tag0*), a tuple, a lambda expression representing a function (e.g. the definition of *msg_to_bytes* on line 10), a constructor application (*Request s*), a function application (*Lib.utf8 s*), a let-expression for sequential composition (line 17) , or a conditional expression written using either pattern matching (line 11) or if-then-else (line 18).

Patterns (*P*) are used for ML-style pattern matching in **match** and **let** expressions. A pattern can be used to check the shape of an expression: whether it is a constant, a tuple (e.g. (*tag*,*sb*) on line 16), or uses a given constructor (*Request s* on line 12). If an expression matches a pattern, then its components are bound to the corresponding variables in the pattern.

In addition, we introduce two new expression constructs (not used in *Msg*) to enable logical specifications within expressions. The expression **assume**$(p\ M)$ states that at this point in the program, the logical predicate p can be assumed to hold at value M. Conversely, the expression **assert**$(p\ M)$ states that at this point in the program, the logical predicate p is expected to hold at value M. During typechecking, these constructs are interpreted as logical annotations that serve as assumptions and verification goals (explained later in this section.) However, they have no effect at run-time—both constructs are implemented as *noop*s in F#. Each predicate symbol p that appears in an expression must be previously declared as a datatype constructor; hence, each **assume** or **assert** is a well-formed F# expression.

The programs in the rest of this tutorial conform to the syntax described here. In addition, they may use predefined values, functions, and types from the F# standard library ("Pervasives"). These include the function = that tests for the (structural) equality of two values, and the type list that has constructors for nil ([]) and cons (::).

2.2 Security Specifications in F7

We define a language for writing logical specifications for F# modules as F7 *interfaces*. The F7 interface language extends F# interfaces, hence every F# interface is an F7 interface, and every F7 interface may be *erased* to a normal F# interface. The full syntax of interfaces is given on the next page.

An F7 interface for a module M is written in a separate file (`m.fs7`) and consists of a sequence of declarations. A declaration (σ) may declare the F7 type of a variable or function (*M.x*, *M.f*), and this type is then expected to be the type of the corresponding value or function definition in the F# module M. A type abbreviation declaration introduces a new type constructor symbol (M.α) as an abbreviation for an F7 type (T). An algebraic type declaration introduces a new type constructor ($M.\alpha$ and a set of term constructors (h_i that takes an argument of F7 type T_i). An abstract type declaration introduces a new type constructor ($M.\alpha$) but leaves its implementation completely hidden. More generally, each type constructor may be parametrized by type arguments (e.g. '*t* list) or by values (e.g. ('*t*; *p*:string) key), but we elide this detail here, and instead we treat parametrized types simply as schemas that are eliminated by inlining. Each value, function, or type declaration may have a visibility qualifier, **public** (default) or **private**: **public** declarations are considered part of the exported interface of the module and mark functions and types that are visible to all other F# modules (including the adversary), whereas **private** declarations are typically used for auxiliary functions and values that are needed for typechecking but are not exported by the module.

We illustrate the syntax by displaying the interface for the *Lib* module used by *Msg*. The interface file `lib.fs7` declares an abstract type bytes that represents byte arrays. It then introduces two logical functions *Utf8* and *Length*, declares F7 types for the functions *utf8*, *iutf8*, *length*, *concat*, and *split*, and introduces three **assume**s.

Syntax of F7 Interfaces (values, formulas, types, declarations)

$M, N ::=$	Values
$(), \mathbf{true}, \mathbf{false}, i, \text{“}s\text{”}$	constants: unit, booleans, integers, strings
x	variable
$(M_1, \ldots, M_n)$	tuple
$h\ M$	constructor
$l\ M$	logical function application
$C ::=$	First-order Logic Formulas
$\mathbf{true} \mid \mathbf{false}$	constants
$M_1 = M_2 \mid M_1 \neq M_2$	comparison
$p\ M$	predicate application
$not\ C \mid C \wedge C \mid C \vee C$	boolean operators
$C \Rightarrow C \mid C \Leftrightarrow C$	implication, if-and-only-if
$!x.C$	universal, first-order quantification (for all)
$?x.C$	existential, first-order quantification (exists)
$T ::=$	Types
$\text{unit}, \text{bool}, \text{int}, \text{string}$	base types
$T\ \mathbf{ref}$	reference
α	type constructor
$x : T\{C\}$	x of type T such that C (scope of x is C)
$x : T_1 \rightarrow T_2$	dependent function type (scope of x is T_2)
$x : T_1 \times T_2$	dependent pair type (scope of x is T_2)
$\pi ::= \mathbf{public} \mid \mathbf{private}$	Visibility qualifier
$\sigma ::=$	Declarations
$\pi\ \mathbf{val}\ x : T$	value or function declaration
$\pi\ \mathbf{type}\ \alpha = T$	type abbreviation
$\pi\ \mathbf{type}\ \alpha = \prod_i (h_i\ \mathbf{of}\ T_i)$	recursive algebraic datatype
$\mathbf{type}\ \alpha$	abstract type
$\mathbf{function\ val}\ l : T_1 \rightarrow T_2$	logical function
$\mathbf{predicate\ val}\ p : T \rightarrow \text{bool}$	logical predicate
$\mathbf{assume}\ C$	assume formula
$\mathbf{assert}\ C$	assert formula
$I ::= \mathbf{module}\ M\ \sigma_1 \ldots \sigma_n$	Interface

A logical function declaration introduces a new function symbol ($M.l$) that may only be used within formulas. The function symbol is initially uninterpreted, but we may add assumptions about its behaviour using an **assume** declaration. For example, our only assumption about *Utf8* is stated in the **assume** on line 14, namely, that *Utf8* is an injective function. When a logical function has more than one argument, they must be written in uncurried form ($l(M_1, \ldots, M_n)$). The *Lib* interface uses a predefined uninterpreted logical function (|) with two arguments that represents byte array concatenation.

A logical predicate declaration (not used in *Lib*) introduces a new predicate symbol ($M.p$) that may be used in formulas in interfaces and **assume** or **assert** expressions.

An **assume** declaration states that a formula (C) is assumed to be true. Such declarations are used to define logical predicates and functions, and to state verification assumptions. For example, the **assume**s on lines 15 and 16 states our assumptions on the relationship between *Length* and byte array concatenation (|); in particular, concate-

Byte array conversions: (`lib.fs7`)

```
1   module Lib
2
3   type bytes
4
5   function val Utf8: string → bytes
6   function val Length: bytes → int
7
8   val utf8: s:string → b:bytes {b = Utf8(s)}
9   val iutf8: b:bytes → s:string {b = Utf8(s)}
10  val length: b:bytes → l:int {Length(b) = l}
11  val concat: b1:bytes → b2:bytes → b:bytes{b = b1|b2}
12  val split: i:int{i > 0} → b:bytes → b1:bytes * b2:bytes{b = b1|b2 ∧ Length(b1) = i}
13
14  assume !x:string, y:string. (Utf8(x)=Utf8(y) ⇒ x = y)
15  assume !x:bytes, y:bytes. Length(x | y) = Length(x) + Length(y)
16  assume !b1,b2,c1,c2. (b1|b2 = c1|c2 ∧ Length(b1)=Length(c1)) ⇒ (b1=c1 ∧ b2=c2)
```

nating n bytes to an array increases its length by n, and if two equal byte arrays are split at the same index, the results are pointwise equal. An **assert** declaration requires that a formula (C) must be entailed by the formulas previously assumed. Such declarations are used to formulate (and verify) lemmas that serve as hints for subsequent steps.

F7 types (T) subsume and extend F# types. Like in F#, an F7 type may be a base type, a mutable reference containing a value of type T (T **ref**), or a type constructor ($M.\alpha$) representing a type abbreviation or algebraic type.

A *refinement type*, written $x : T\{C\}$, represents a value M of type T such that $C\{M/x\}$ holds. For example, the type *b*:bytes $\{b = Utf8(s)\}$ (line 8) represents all byte arrays M that are equal to $Utf8(s)$.

A dependent pair type, written $x : T_1 \times T_2$, represents a pair of values (M,N) such that M has type T_1 and N has type $T_2\{x/M\}$. In particular, we can use dependent pair types to write dependent tuples of the form $x_1 : T_1 \times \ldots \times x_n : T_n$. For example, the type *b1*:bytes * *b2*:bytes$\{b = b1|b2 \wedge Length(b1)= i\}$ (line 12) represents a pair of byte arrays (*M*,*N*) where the refinement type of the second component refers to the first.

A dependent function type, written $x : T_1 \rightarrow T_2$, represents a function that takes an argument M of type T_1 and returns a result of type $T_2\{M/x\}$. A function type is often written in the form $x : T_1\{C_1\} \rightarrow y : T_2\{C_2\}$; here, the formula C_1 is called the precondition and C_2 is called the post-condition. For example, the type of *split* (line 12) is a function type that takes two arguments, an integer *i* and byte array *b*, and returns a pair of byte arrays (*b1*,*b2*); its precondition is that *i* must be greater than 0; its post-condition is that *b* is the concatenation of *b1* and *b2* and that the length of *b1* is *i*.

Formulas (C) appear in refinement types and in **assume** and **assert** declarations. Formulas are written in an untyped first-order logic with uninterpreted function and predicate symbols. They may use standard operators for conjunction, disjunction, implication, and universal and existential quantification. For example, the **assume** on line 16 uses equality, conjunction, implication, and universal quantification. Each predicate or

logical function has a fixed arity and must be declared before it can be used. For example, the predefined predicates for equality and inequality have arity 2. The behaviour of predicates can be defined by assuming formulas of the form $\forall x.p\ x \Leftrightarrow C$.

Formulas speak about values (M). A value may be a constant such as a boolean, an integer, or a string (interpreted as a 0-ary function symbol). It may also be a constructed term (hM), where the constructor h is treated as a 1-ary function symbol that is one-to-one and invertible. It may be an n-tuple ($(M_1,\ldots,M_n)$), which is interpreted as the application of an n-ary tuple constructor ($Tuple_n(M_1,\ldots,M_n)$). Finally, a value may be the application of a logical function symbol ($l\ M$), such as $Length(b)$.

2.3 Modular Verification by Typechecking

We consider programs of the form:

$$L_1\ \cdots\ L_m\ P_1\ \cdots\ P_n\ \cdot\ O$$

where the modules $L_1,\ldots,L_m$ are trusted libraries that are assumed to behave correctly, the modules $P_1,\ldots,P_n$ are program modules that we wish to verify, and the module O represents an arbitrary adversary that does not contain any **assume**s or **assert**s. The adversary O is given access to the library and program modules through their exported (public) interfaces and hence, it may read any public variable and call any public function (any number of times). The adversary does not have to be well-typed in F7, but it must obey the F# types of the exported interface. Our verification goal is *semantic safety*: in every run of the program, every asserted formula logically follows from previously assumed formulas. In particular, if the **assert** expressions in a program represents security goals, then these security goals should be satisfied in all runs of the program for all adversaries O.

Our proof method is typechecking: we verify that each F# module in the program satisfies its typed F7 interface. Each judgment of the F7 type system is given relative to an *environment*, E, which is a sequence $\mu_1,\ldots,\mu_n$, where each μ_i may be a value or type declaration (σ), or *subtype assumption* $\alpha <: \alpha'$. The two main judgments are *subtyping*, $E \vdash T <: U$, and *type assignment*, $E \vdash e : T$. Gordon and Fournet (2010) give the full typing rules for these judgments. Informally, F7 adds the formula C to the current logical environment when processing **assume** C, and conversely checks that formula C is provable when processing **assert** C. Similarly, before processing the body of a function with expected type $x\!:\!T_1\{C_1\} \to y : T_2\{C_2\}$, F7 adds the precondition C_1 to the current logical environment; after processing the body, it verifies that the post-condition C_2 is provable. F7 implements these typing rules to check type assignment for modules: $I_1,\ldots,I_n \vdash P \rightsquigarrow I$, that is, given modules with interfaces $I_1,\ldots,I_n$, the module P satisfies the F7 interface I. F7 relies on various type inference algorithms, and calls out to an SMT solver Z3 to handle the logical goals that arise when checking formulas.

Library modules (such as *Lib*) are typically modules that rely on the underlying operating system or those for which we do not have the full source code. For each library module L_i, we specify its behaviour as an F7 interface I_i^L and provide an idealized implementation L_i' that satisfies this interface. For each program module P_i we define

an F7 interface I_i^P and a public interface I_i^O. We then modularly verify each library and program module in sequence; for each library module L_i we verify by typechecking that

$$I_1^L, \ldots, I_{i-1}^L \vdash L_i' \rightsquigarrow I_i^L$$

for each module P_i we verify by typechecking that

$$I_1^L, \ldots, I_m^L, I_1^P, \ldots, I_{i-1}^P \vdash P_i \rightsquigarrow I_i^P <: I_i^O$$

By composing these typechecking steps, we establish:

$$\vdash L_1' \ \cdots \ L_m' \ P_1 \ \cdots \ P_n \rightsquigarrow I_1^L \ \cdots I_m^L \ I_1^O \ \cdots \ I_n^O$$

We then use the type-soundness theorem of F7 to obtain

Theorem 1 (Semantic Safety by Typing) *For any opponent O,*
if $I_1^L, \ldots, I_m^L, I_1^O, \ldots, I_n^O \vdash O$: unit*, then* $L_1' \ \cdots \ L_m' \ P_1 \ \cdots \ P_n \ \cdot \ O$ *is semantically safe.*

2.4 Example: Verifying *Lib* · *Msg*

We illustrate our verification methodology on the program composed of `lib.fs` and `msg.fs`. Using the F# compiler, this program can be compiled to a library that may be used to serialize (and deserialize) messages as byte arrays:

```
fsc lib.fs msg.fs -a -o msg.dll
```

The file `lib.fs` implements our library module *Lib* by calling functions in the .NET framework:

Byte array conversions: (defined in `lib.fs`)

type bytes = *byte array*
let *utf8* (*x*:string) : bytes = *System.Text.Encoding.UTF8.GetBytes x*
let *iutf8* (*x*:bytes) : string = *System.Text.Encoding.UTF8.GetString x*
let *length* (*x*:bytes) = *x.Length*
let *concat* (*x0*:bytes) (*x1*:bytes) : bytes = *Array.append x0 x1*

This code is not verified, since the underlying code for `System.Text` is not in F#, so we just trust that this library meets the assumptions documented in the F7 interface for *Lib* (as shown above).

We write an interface file `msg.fs7` for the program module *Msg* as follows:

Serializing messages as byte arrays: (`msg.fs7`)

```
module Msg

type msg =
  | Request of string
  | Response of string

val tag0 : Lib.bytes
val tag1 : Lib.bytes

predicate val Serialized: msg * Lib.bytes → bool
assume !m,b. Serialized(m,b) ⇔
  ((!s. m = Request(s) ⇒ b = tag0 | Lib.Utf8(s)) ∧
   (!s. m = Response(s) ⇒ b = tag1 | Lib.Utf8(s)))

val msg_to_bytes: m:msg → b:Lib.bytes{Serialized(m,b)}
val bytes_to_msg: b:Lib.bytes → m:msg{Serialized(m,b)}
```

The interface file declares the type *msg* (as in module *Msg*), it declares types for the values *tag0* and *tag1*, and gives types for the functions *msg_to_bytes* and *bytes_to_msg*, using a new predicate *Serialized*. These types represent our correctness specification for the program module.

To verify our program, we execute F7 as follows:

```
> f7 lib.fs7 msg.fs7 msg.fs
```

This verifies that:

$$I_{Lib} \vdash Msg \rightsquigarrow I_{Msg}$$

(where we use the notation I_M to refer to the interface of a module M. For our module *Msg*, typechecking succeeds; the module satisfies its interface. To illustrate a typechecking error, let us introduce an error in *Msg.msg_to_bytes* as follows

```
let msg_to_bytes = fun m →
  match m with
   | Request s → Lib.concat tag0 (Lib.utf8 s)
   | Response s → Lib.concat tag0 (Lib.utf8 s)
```

On the last line, we incorrectly use *tag0* instead of *tag1*. Now, when we try to verify our program, it generates an error:

```
> f7 lib.fs7 msg.fs7 msg.fs
    msg.fs(13,19): Error: Query fails...
    ...cannot prove Msg.Serialized(m, l343)
```

Hence, F7 could not prove the post-condition of *msg_to_bytes*. F7 also provides a full logical trace that lists all logical formulas available as hypotheses in the failed proof attempt.

3 An Authenticated RPC

We consider a sample protocol intended to authenticate remote procedure calls (RPC) over a TCP connection. In this section, we informally discuss the security of this protocol and identify a series of underlying assumptions. In Section 4, we then explain how to formalize these assumptions, In Section 5, we show how to verify our implementation by typing against a symbolic cryptographic library. Finally, in Section 6, we show how to verify the same implementation against a concrete cryptographic library, under standard computational assumptions.

3.1 Informal Description

We have a population of principals, represented as concrete strings ranged over by a and b, that intend to perform various remote procedure calls (RPCs) over a public network. The RPCs consist of requests and responses, both also represented as strings. The security goals of our RPC protocol are that (1) whenever a principal b accepts a request message s from a, principal a has indeed sent the message to b and, conversely, (2) whenever a accepts a response message t from b, principal b has indeed sent the message in response to a matching request from a.

To this end, the protocol uses message authentication codes (MACs) computed as keyed hashes, such that each symmetric MAC key k_{ab} is associated with (and known to) the pair of principals a and b. Our protocol may be informally described as follows.

An Authenticated RPC Protocol:

1. $a \rightarrow b : \ \mathit{utf8}\ s \mid (\mathit{MAC}\ k_{ab}\ (\mathit{request}\ s))$
2. $b \rightarrow a : \ \mathit{utf8}\ t \mid (\mathit{MAC}\ k_{ab}\ (\mathit{response}\ s\ t))$

In the protocol narration, each line indicates an intended communication of data from one principal to another. This data is built using five functions.

- *utf8*, of type string $\rightarrow$ bytes, is a function that marshals strings such as s and t into byte arrays (the message payloads) using a standard character encoding.
- '|', of type bytes $\rightarrow$ bytes $\rightarrow$ bytes, concatenates the message parts.
- *request*, of type string $\rightarrow$ bytes and *response*, of type string $\rightarrow$ string $\rightarrow$ bytes, build message digests (the authenticated values). These functions may for instance be implemented as tagged concatenations of their utf8-encoded arguments.
- *MAC*, of type key $\rightarrow$ bytes $\rightarrow$ bytes computes keyed cryptographic hashes of these message digests (their MACs); this function may be implemented for instance using the HMACSHA1 algorithm.

3.2 Security Considerations

We consider systems in which there are multiple concurrent RPCs between any principals a and b of the population. The adversary controls the network and some actions of the participants. Some keys may also become compromised, that is, fall under the control of the adversary. Intuitively, the security of the protocol depends on the following assumptions:

(1) The function *MAC* is cryptographically secure, so that MACs cannot be forged without knowing their key. We will carefully detail this security assumption, both symbolically and computationally.
(2) The principals a and b are not compromised—otherwise the adversary may just obtain the key k_{ab} and then form MACs for arbitrary requests and responses.
(3) The formatting functions *request* and *response* are injective and their ranges are disjoint—otherwise, an adversary may for instance replace the first message payload with *utf8* s' for some $s' \neq s$ such that *request* $s' =$ *request* s and thus get s' accepted instead of s, or it may use a request MAC to fake a response message.
(4) The key k_{ab} is a genuine MAC key shared between a and b, used exclusively for building and checking MACs for requests from a to b and responses from b to a—otherwise, for instance, if b also uses k_{ab} for authenticating requests from b to a, it would accept its own reflected messages as valid requests from a.

These assumptions can be precisely expressed (and verified) as *program invariants* of the protocol implementation. Moreover, the abstract specifications of *MAC*, *request*, and *response* given above should suffice to establish security of our protocol, irrespective of their implementation details.

Exercise 1. Write a protocol narration showing that the protocol is *not* secure with the definition of *request* and *response* given below. How could we define these two functions securely?

let *request s* = *utf8* `"Request"` | *utf8 s*
let *response s t* = *utf8* `"Response"` | *utf8 s* | *utf8 t*

Exercise 2. Our sample protocol does not offer much protection against replay attacks. How could we prevent the replay of request messages?

3.3 Logical Specification: Adding Events and Assertions

Next, we introduce logical events to specify the trace properties of a system that runs our protocol: we use event predicates to record the main steps of each instance of the protocol, to record the association between keys and principals, and to record principal compromise. These events are parametrized with "high-level" strings, recording the principal, requests, and responses actually passed to the application that use the protocol, rather than their low-level encodings as bytes.

To mark an event in code, we assume a corresponding logical fact:

- $\mathit{Request}(a,b,s)$ before a sends message 1 to b with request s;
- $\mathit{Response}(a,b,s,t)$ before b sends message 2 to a with response t to the request s;
- $\mathit{KeyAB}(k,a,b)$ before issuing a key k associated with a and b;
- $\mathit{Corrupt}(a)$ before leaking any key associated with a.

(Recall that logical events are used only for specification purposes. Hence, $\mathit{Corrupt}(a)$ does not presume some dynamic intrusion detection mechanism; instead, it abstractly records that a is compromised, so that we can condition our formal expectations.)

To state an intended security goal in terms of these events, we assert that a logical formula always holds at a given location in our code, in any system configuration, and despite the presence of an active adversary. In our protocol, we assert:

- *Request*(*a*,*b*,*s*) ∨ *Corrupt*(*a*) ∨ *Corrupt*(*b*)
  after *b* accepts message 1 as a request *s* from *a*;
- (*Request*(*a*,*b*,*s*) ∧ *Response*(*a*,*b*,*s*,*t*)) ∨ *Corrupt*(*a*) ∨ *Corrupt*(*b*)
  after *a* accepts message 2 as a response *t* from *b* to its request *s*.

In these formulas, the disjunctions account for the potential compromise of (at least) one of the two principals with access to the MAC key—see assumption (2) in our informal security considerations. In particular, if we consider a simpler, weaker attacker model such that the principals *a* and *b* are never compromised, then *Corrupt*(*a*)∨*Corrupt*(*b*) is always false and the asserted formulas are in direct correspondence with the assumed events *Request*(*a*,*b*,*s*) and *Response*(*a*,*b*,*s*,*t*).

3.4 A Concrete Library for MACs

We now describe our protocol implementation, which consists of three F# modules: *Mac*, a library for MACs, *Format*, a module for message formatting, and *RPC*, a module for the rest of the protocol code. (Placing the formatting functions *request* and *response* in a separate module is convenient to illustrate modular programming and verification.) Except for protocol narrations, all the code displayed in this tutorial is extracted from F7 interfaces and F# implementations that have been typechecked.

We begin with our cryptographic library. Message authentication codes (MACs) provide integrity protection based on keyed cryptographic hashes. We define an F# module that implements this functionality. We first give its plain F# interface:

```
type key
type text = bytes
type mac = bytes
val macsize: int
val keysize: int
val GEN: unit → key
val MAC: key → text → mac
val VERIFY: key → text → mac → bool
val LEAK: key → bytes
```

The interface declares type abbreviations for keys, texts, and MACs. These types are just type aliases for 'bytes', the type of concrete byte arrays, used for clarity in other declarations. The interface also declares symbolic constants for the sizes (in bytes) of MACs and keys, and four functions: to generate a fresh key; to produce a MAC given a key and a text; to verify whether a MAC is valid given a key and a text; and to serialize a key into bytes. This last function is named *LEAK*, as we will use it to model key compromise. We give below a concrete implementation of that interface, based on the .NET cryptographic libraries, which we use for running our protocol.

```
open System.Security.Cryptography
let keysize = 16 (* 128 bits *)
let macsize = 20 (* 160 bits *)
type key = bytes
let GEN () = randomBytes keysize
let MAC k (t:text) = (new HMACSHA1(k)).ComputeHash t
let VERIFY k t sv = (MAC k t = sv)
let LEAK (k:key) = k
```

(For brevity, we often omit repeated declarations in code excerpts, such as **type** text = bytes above; we refer to the code online for the complete source files.) This F# code

implements the interface above; it sets sizes for keys and MACs, samples concrete random bytes as keys, and just performs system calls to standard algorithms (Krawczyk et al. 1997; Eastlake and Jones 2001).

As with all practically deployed symmetric primitives, there is no formal security guarantee, and the choice of algorithms is expected to evolve over time. On the contrary, since MACs may be much smaller than texts, there are many texts with the same MACs; we just hope that collisions are hard to find.

3.5 Implementing the RPC Protocol

We give on the next page an F# implementation of our RPC protocol. This code relies on general libraries *Pi* (for specifications) and *Net* (for basic TCP networking), on the *Mac* module described above, and on the *Format* module, whose concrete implementation is deferred till Section 6.

Compared to the protocol narration, the code details message processing, and in particular the series of checks performed when receiving messages. For example, upon receiving a request, the code that defines *server* first extracts *s* from its encoded payload by calling *iutf8*, then it verifies that the received MAC matches the MAC recomputed from *k* and *s*.

The code assumes events that mark the generation of a key for our protocol and the intents to send a request from *a* to *b* or a response from *b* to *a*. The code asserts two properties, after receiving a request or a response, and accepting it as genuine.

3.6 Modelling Active Adversaries

We model an opponent as an arbitrary program with access to a given *adversary interface* that reflects all its (potential) capabilities. Thus, our opponent has access to the network (modelling an active adversary), to cryptography (modelling access to the MAC algorithms), and to a protocol-specific interface that yields four capabilities: to generate a MAC key for a pair of principals; to demand the MAC keys of a principal (which becomes *Corrupt*); to run the client with some payload, and to run the server. This interface is

```
val keygen: principal → principal → unit
val corrupt: principal → bytes list
val client: principal → principal → string
val server: principal → principal → (string → string) → unit
```

It exports four functions similar to those defined by *RPC*, except that the keys are omitted Instead, their implementation maintains a private, global, mutable table of all keys that have been allocated so far, and uses table lookups, calls to the functions of *RPC*, and table updates.

```
let keys = ref []
let keygen a b = let k = RPC.keygen() in keys:=((a,b),k)::!keys
let corrupt x = List.fold (fun ((a,b),k) ks → if x=a || x=b then LEAK k::ks else ks) !keys
let client a b s = let k = List.assoc !keys (a,b) in client a b k s
val server a b f = let k = List.assoc !keys (a,b) in server a b k f
```

F# Implementation for the Authenticated RPC Protocol: (`rpc.fs`)

```
module RPC
open Pi
open Lib
open Mac
open Format

type principal = string
type fact =
  | Request of principal * principal * string
  | Response of principal * principal * string * string
  | KeyAB of key * principal * principal
  | Corrupt of principal
  | RecvRequest of principal * principal * string
  | RecvResponse of principal * principal * string * string
let keygen (a:principal) (b:principal) =
  let k = GEN() in
  assume(KeyAB(k,a,b)); k

let corrupt (a:principal) (b:principal) k =
  assume(Corrupt(a)); LEAK k

let client (a:principal) (b:principal) k s =
  assume (Request(a,b,s));
  let c = Net.connect p
  let msg1 = concat (utf8 s) (MAC k (request s))
  Net.send c msg1
  let msg2 = recv()
  let (v,m') = split macsize msg2
  let t = iutf8 v
  if VERIFY k (response s t) m' then
    assert (RecvResponse(a,b,s,t)); Some(t)
  else None

let server (a:principal) (b:principal) k service =
  let msg1 = Net.listen p in
  let (v,m) = split macsize msg1 in
  let s = iutf8 v in
  if VERIFY k (request s) m then
    assert (RecvRequest(a,b,s));
    let t = service s in
    assume (Response(a,b,s,t))
    let msg2 = concat (utf8 t) (MAC k (response s t))
    send msg2
```

Formally, the opponent ranges over arbitrary F# code well-typed against an interface that includes (at least) the declarations below. (Demanding that the opponent be well-typed is innocuous as long as the interface only operates on plain types such as bytes.)

3.7 Running the RPC Protocol

Experimentally, we test that our protocol code runs correctly, at least when compiled with a *Net* library and an "adversary" that implements a reliable network and triggers a single test query. We build our executable using a command of the form

```
> fsc lib.fs mac.fs format.fs rpc.fs test.fs -o rpc.exe
```

and the messages exchanged over TCP are:

```
> ./rpc.exe
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYS0GGtOtPm...} (23 bytes)
Received Response 4
```

4 Logical Refinements for the RPC Protocol

We now develop more precise typed interfaces that embed logical refinements to support our verification effort. The interfaces presented is this section are used both for symbolic and computational verification.

4.1 Refined Interface for Formatting the Authenticated Digests

We introduce two auxiliary predicates for the payload formats: *Requested* and *Responded* are the (typechecked) postconditions of the functions *request* and *response*. Hence, the *Format* module used by the protocol implementation exports the following interface.

Interface to Format Modules (`format.fs7`):

module *Format*
open *Lib*
predicate val *Requested*: bytes $*$ string $\rightarrow$ bool
predicate val *Responded*: bytes $*$ string $*$ string $\rightarrow$ bool
val *request*: *s*:string $\rightarrow$ *b*:bytes $\{$*Requested*(*b*,*s*)$\}$
val *response*: *s*:string $\rightarrow$ *t*: string $\rightarrow$ *b*:bytes $\{$*Responded*(*b*,*s*,*t*)$\}$
assert!*v*,*v*',*s*,*s*',*t*'. (*Requested*(*v*,*s*) $\wedge$ *Responded*(*v*',*s*',*t*')) $\Rightarrow$ ($v \neq v$')
assert!*v*,*v*',*s*,*s*'. (*Requested*(*v*,*s*) $\wedge$ *Requested*(*v*',*s*') $\wedge$ $v = v$') $\Rightarrow$ ($s = s$')
assert!*v*,*s*,*s*',*t*,*t*'. (*Responded*(*v*,*s*,*t*) $\wedge$ *Responded*(*v*,*s*',*t*')) $\Rightarrow$ ($s = s$' $\wedge$ $t = t$')

Typechecking involves the automatic verification that our formatting functions are injective and have disjoint ranges, as informally explained in informal assumption (3). Verification is triggered by asserting the formulas marked with **assert**, so that Z3 proves them. For verifying the rest of the protocol, we can hide the definition of *Requested* and *Responded* and rely only on these three derived logical properties, which confirms that the security of our protocol depends only on these properties, rather than a specific digest format.

4.2 Refined, Ideal Interface for Cryptographic MACs

To capture the intended properties of MACs, we rely on another, refined *ideal interface* for the *Mac* module, as follows:

type bytespub = bytes
type key
type text = bytes
type mac = b:bytes$\{$*Length*(b) = *macsize*$\}$
predicate val *Msg*: key $*$ text $\rightarrow$bool
val *GEN*: unit $\rightarrow$key
val *MAC*: k:key $\rightarrow$ t:text$\{$*Msg*(k,t)$\}$ $\rightarrow$mac
val *VERIFY*: k:key $\rightarrow$ t:text $\rightarrow$ m:mac $\rightarrow$ v:bool$\{v$=**true** $\Rightarrow$ *Msg*(k,t)$\}$
val *LEAK*: k:key$\{!t.$ *Msg*(k,t)$\}$ $\rightarrow$ b:bytes $\{$*Length*(b) = *keysize*$\}$

This refined interface is designed for protocol verification, and is similar to those used for typing symbolic models of cryptography (Bhargavan et al. 2010). It declares the type of keys as abstract, thus preventing accidental key leakage in data sent on a public network or passed to *MAC*.

To support authentication properties, the interface introduces a logical predicate on keys and texts, *Msg*(k,t), to indicate that t is an authentic message MACed with key k. (In F7 syntax, predicates are declared as constructors.) This predicate occurs in the types of *MAC* and *VERIFY*, as a precondition for MACing and as a post-condition of successful verification. The interpretation of *Msg*(k,t) is protocol-dependent: as illustrated next for the RPC protocol, each protocol defines *Msg* according to the properties it wishes to authenticate using MACs, possibly giving a different interpretations to each key. Since our interface must be sound for any logical definition of *Msg*, this entails that, until the key is leaked, calls to *VERIFY* succeeds *at most* for texts previously passed as arguments to *MAC*, thus excluding the possibility of forging a MAC for any other text.

The precondition of *LEAK* accounts for (well-typed) dynamic key compromise: the protocol may call *LEAK* to access the actual key bytes at any point, while other parts of the protocol still rely on the postcondition of *VERIFY*. Thus, the precondition $!t$, *Msg*(k,t) demands that, before leaking the key, *Msg*(k,t) holds for all texts. (In F7 syntax, $!t$ is logical universal quantification over t.) The function *LEAK* can be used to model the potential corruption of principals by leaking their keys to the adversary; see Bhargavan et al. (2009a, 2010) for protocols verified by typing despite partial key compromise.

Taking advantage of refinements, the interface also enforces the consistency of sizes for MACs and key bytes. (In F7, *Length* is a logical library function.) Thus, for instance, typechecking would catch parsing errors leading to a call to *VERIFY* with a truncated MACs, as its argument would not match the refined type mac.

Intuitively, the ideal interface is too good to be true: a simple information-theoretic argument shows that any implementation such that (1) MACs are shorter than texts and (2) *MAC* and *VERIFY* communicate only through MACs must be unsafe. (In particular, we do *not* have $\vdash C \rightsquigarrow I$.) In the next two sections, we will show how to formally meet the properties exported by the ideal interface, first symbolically, by implementing MACs using abstracts terms instead of concrete byte arrays, then computationally, by implementing an intermediate error-correcting ideal functionality for MACs.

4.3 Instantiating the MAC Library to the RPC Protocol

We are now ready to define the predicate *Msg* for the RPC protocol, by relating the format of the bytes that are cryptographically authenticated to their interpretation in terms of authentic requests and responses.

Usage of Keys: (`rpc.fs7`)

definition
!*a*,*b*,*k*. *KeyAB*(*k*,*a*,*b*) ⇒
(!*m*. *Msg*(*k*,*m*) ⇔
Corrupt(*a*) ∨ *Corrupt*(*b*) ∨
(?*s*. *Requested*(*m*,*s*) ∧ *Request*(*a*,*b*,*s*)) ∨
(?*s*,*t*. *Responded*(*m*,*s*,*t*) ∧ *Response*(*a*,*b*,*s*,*t*)))

For each key generated for the protocol, the definition gives four valid interpretations: one of the two parties is corrupted, or this is a correctly-formatted request, or this is a correctly-formatted response.

This definition is not in itself trusted. It is included in the F7 interface for RPC and used to typecheck its code. If the definition was too strong, then we would get a typing error in the code that calls *MAC* and needs to prove *Msg*(*k*,*t*) as a precondition. If the definition was too weak, then we would get a typing error at the assertions, as the post-condition of *VERIFY* would not suffice to prove the asserted formula.

Exercise 3. Explain how to model the re-use of keys in both directions, so that a key *k* shared between *a* and *b* may be used also for authenticating requests from *b* to *a* and responses from *a* to *b*?

What is the resulting definition of predicate *Msg*(*k*,*m*)? Is the resulting sample protocol well-typed? Is it secure? Otherwise, describe how to fix it.

Exercise 4. Explain how to model sessions that consist of a sequence of requests and responses? What is the resulting definition of predicate *Msg*(*k*,*m*)?

4.4 Towards Security Theorems for our Sample Protocol

Before presenting our verified cryptographic libraries, we outline the general form of our security theorems for the RPC protocol, given in the next two sections. We consider the runtime safety of programs of the form

$$Lib \cdot MAC \cdot Format \cdot RPC \cdot Adv$$

where

– *Lib* is a standard library;
– *MAC* is our cryptographic library for MACs;
– *RPC* is our protocol code; and
– *Adv* is an active adversary, ranging over programs well-typed against the adversary interface. These programs cannot directly assume or assert logical properties, although they can cause *RPC* and *MAC* to do so by calling them through the adversary interface. Without loss of generality, we assume that *Adv* returns (): unit.

Our protocol is deemed secure when all runs of programs of the form above are safe, that is, all their assertions follow from their assumptions. Informally, this excludes any authentication and correlation attack against our protocol using the capabilities granted by the adversary interface.

To prove that $Lib \cdot MAC \cdot Format \cdot RPC \cdot Adv$ is safe, we will mostly rely on typing: we prove that

$$\vdash Lib \cdot MAC \cdot Format \cdot RPC \cdot Adv : unit \tag{1}$$

then conclude by type safety. More precisely, we use F7 to perform the following tasks:

(1) typecheck $\vdash Lib \rightsquigarrow I_{Lib}$ (possibly assuming properties of primitive functions);
(2) typecheck $I_{Lib} \vdash MAC \rightsquigarrow I_{MAC}$;
(3) typecheck $I_{Lib} \vdash Format \rightsquigarrow I_{Format}$;
(4) typecheck $I_{Lib}, I_{MAC}, I_{Format} \vdash RPC \rightsquigarrow I_{RPC}$;
(5) check the subtyping relation $I_{Lib}, I_{MAC}, I_{RPC} <: I_{adv}$.

If we further suppose that $I_{adv} \vdash Adv : unit$, then, by composing all these typing judgments, we obtain the whole-program typing (1).

These tasks are independent, so the actual verification effort for RPC using existing cryptographic libraries essentially amounts to typechecking the implementations of *Format* and *RPC*.

5 Symbolic Soundness

In the approach to cryptographic verification pioneered by Dolev and Yao (1983), we treat byte arrays as if they correspond to symbolic algebraic terms, with some idealized properties. In our simple setting, the following algebra suffices for describing byte arrays.

Algebra of Terms: (`mac-symb.fs7`)

```
type term =
  | Key of Pi.name
  | Utf8 of string
  | ByteTag of int * term
  | Pair of term * term
  | Mac of term * term
```

A term $Key(k)$ represents the bytes of a key k, where k itself is a *name* in the sense of the pi calculus (Milner 1999). We use a primitive type *Pi.name*, whose values are atoms in the style of pi calculus names. The only operations on names are to test for equality and to freshly generate new names—as in the restriction operator of the pi calculus. It is impossible for an observer, such as the attacker, to guess in advance the value of a fresh name. The use of pi calculus names and restriction to model cryptographic keys and key generation was introduced by Abadi and Gordon (1999).

A term $Utf8(s)$ represents the bytes encoding the string s. A term $ByteTag(i,t)$ represents the byte i, followed by the bytes represented by the term t. A term $Pair(t_1,t_2)$ represents the concatenation of the bytes represented by the terms t_1 and t_2, preceded by the number of bytes of t_1, so that the whole may be unparsed into t_1 and t_2.

Finally, a term $Mac(k,m)$ represents the MAC of message m with key k.

Given the term algebra, we provide the following symbolic implementations of formatting. Each of the primitive formatting functions (*utf8*, *tag*, and *pair*) has a partial inverse, named with an *i*, which attempts to reverse the formatting.

F# Symbolic Implementation for Format Primitives: (`mac-symb.fs`)

```
let utf8 x = Utf8(x)
let iutf8 b = match b with | Utf8 s -> s | _ -> failwith "iutf8 failed"
let tag t x = ByteTag(t,x)
let itag tx = match tx with | ByteTag(t,x) -> t,x | _ -> failwith "itag failed"
let pair x y = Pair(x,y)
let ipair (xs:bytes) = match xs with | Pair(x,y) -> (x,y) | _ -> failwith "ipair failed"
```

Next, we show the symbolic implementation of MACs. The *GEN* function calls the library function *Pi.name* to create a new name n so that the fresh key takes the form $k = Key(n)$. We assume the predicate $MKey(k)$ to record that k is a fresh MAC key generated by a protocol participant. (In principle, there may be many different sorts of key.)

F# Symbolic Implementation for MACs: (`mac-symb.fs`)

```
let GEN() = let k = Key(Pi.name "MAC Key") in assume (MKey(k)); k
let MAC k t = Mac(k,t)
let VERIFY k t m = (m = Mac(k,t))
let LEAK k = k: bytespub
```

We are not concerned with all possible values of type *term*, but only with those that preserve certain cryptographic invariants (Bhargavan et al. 2010). For example, we only consider a term $Mac(k,m)$ if one of two conditions holds: either (1) k is a MAC key and m is an authentic message to be MACed with k, or (2) both k and m are public terms known to the attacker. To state these invariants we rely on predicates with the following intended meanings:

- $Bytes(t)$ holds when bytes t appear in the protocol run;
- $Pub(t)$ holds when the bytes t may be known to the opponent;

We have that $Pub(t)$ implies $Bytes(t)$, so that $Bytes(t)$ holds of all terms t that may arise, including secret terms unknown to the adversary, while $Pub(t)$ holds of the terms t that may actually flow to the adversary. We define these predicates to be the least relations closed under the inductive rules assumed as formulas below. The four rules with $Mac(k,t)$ in their conclusion formalize the invariant on MACs stated above.

The Cryptographic Invariants Pub and Bytes: (`mac-symb.fs7`**)**

assume !*k*. *MKey*(*k*) ⇒ *Bytes*(*k*)
assume !*k*. *MKey*(*k*) ∧ (!*b*. *Msg*(*k*,*b*)) ⇒ *Pub*(*k*)

assume !*i*:int,*b*. *Bytes*(*b*) ⇒ *Bytes*(*ByteTag*(*i*,*b*))
assume !*i*:int,*b*. *Pub*(*b*) ⇒ *Pub*(*ByteTag*(*i*,*b*))

assume !*b1*,*b2*. *Bytes*(*b1*) ∧ *Bytes*(*b2*) ⇒ *Bytes*(*Pair*(*b1*,*b2*))
assume !*b1*,*b2*. *Pub*(*b1*) ∧ *Pub*(*b2*) ⇒ *Pub*(*Pair*(*b1*,*b2*))

assume !*s*:string. *Bytes*(*Utf8*(*s*))
assume !*s*:string. *Pub*(*Utf8*(*s*))

assume !*k*,*t*. *Pub*(*k*) ∧ *Bytes*(*t*) ⇒ *Bytes*(*Mac*(*k*,*t*))
assume !*k*,*t*. *Pub*(*k*) ∧ *Pub*(*t*) ⇒ *Pub*(*Mac*(*k*,*t*))
assume !*k*,*t*. *MKey*(*k*) ∧ *Bytes*(*t*) ∧ *Msg*(*k*,*t*) ⇒ *Bytes*(*Mac*(*k*,*t*))
assume !*k*,*t*. *MKey*(*k*) ∧ *Pub*(*t*) ∧ *Msg*(*k*,*t*) ⇒ *Pub*(*Mac*(*k*,*t*))

We assume the following facts about *Pub* and *Bytes*, to help during typechecking. These formulas are theorems that follow by induction on the derivation of the predicates.

Derived Theorems: (`mac-symb.fs7`**)**

assume !*k*,*t*. *MKey*(*k*) ∧ *Bytes*(*t*) ∧ *Bytes*(*Mac*(*k*,*t*)) ⇒ *Msg*(*k*,*t*)
assume !*k*,*t*. *MKey*(*k*) ∧ *Pub*(*t*) ∧ *Pub*(*k*) ⇒ *Msg*(*k*,*t*)
assume !*i*,*b*. *Bytes*(*ByteTag*(*i*,*b*)) ⇒ *Bytes*(*b*)
assume !*i*,*b*. *Pub*(*ByteTag*(*i*,*b*)) ⇒ *Pub*(*b*)
assume !*b1*,*b2*. *Bytes*(*Pair*(*b1*,*b2*)) ⇒ (*Bytes*(*b1*) ∧ *Bytes*(*b2*))
assume !*b1*,*b2*. *Pub*(*Pair*(*b1*,*b2*)) ⇒ (*Pub*(*b1*) ∧ *Pub*(*b2*))
assume !*t*. *Pub*(*t*) ⇒ *Bytes*(*t*)

Finally, given the term algebra, and our inductively defined predicates, we interpret the abstract types of our protocol with the following refinement types.

Types in the Symbolic Model: (`mac-symb.fs7`**)**

type bytes = *t*:*term* {*Bytes*(*t*)}
type key = *k*:bytes {*MKey*(*k*)}
type bytespub = *t*:bytes {*Pub*(*t*)}
type text = bytes
type mac = bytes

Given these definitions, we can use F7 to verify that the symbolic implementations of formatting and MACs satisfy the following two interfaces. satisfy the following interface.

F7 Interface for Formatting Primitives: (`mac-symb.fs7`**)**

val *utf8*: *x*:string → *y*:bytespub {*y*=*Utf8*(*x*)}
val *iutf8*: *x*:bytespub → *y*:string
val *tag*: *t*:int → *x*:bytes → *tx*:bytes {*tx*=*ByteTag*(*t*,*x*)}
val *itag*: *tx*:bytes → *t*:int ∗ *x*:bytes {*tx*=*ByteTag*(*t*,*x*)}
val *pair*: *x*:bytes → *y*:bytes → *xs*:bytes {*xs*=*Pair*(*x*,*y*)}
val *ipair*: *xs*:bytes → *x*:bytes ∗ *y*:bytes {*xs*=*Pair*(*x*,*y*)}

F7 Interface for MACs: (`mac-symb.fs7`**)**

val *GEN*: unit → *k*:key
val *MAC*:
  k:*term* → *t*:text {(*MKey*(*k*) ∧ *Msg*(*k*,*t*)) ∨ (*Pub*(*k*) ∧ *Pub*(*t*))} → *m*:mac {(*Pub*(*t*) ⇒ *Pub*(*m*))}
val *VERIFY*:
  k:*term* {*MKey*(*k*) ∨ *Pub*(*k*)} → *t*:text → *m*:mac → *v*:bool {(*v*=**true** ∧ *MKey*(*k*)) ⇒ *Msg*(*k*,*t*)}
val *LEAK*: *k*:key{!*t*. *Msg*(*k*,*t*)} → *r*: bytespub {*r*=*k*}

5.1 Verifying Symbolic Message Formats

We now symbolically define our two formatting functions and verify them against their refined interface, `format.fs7` listed in Section 4.1. To this end, we logically define its predicates in terms of the symbolic algebra, as follows.

definition !*b*,*s*. *Requested*(*b*,*s*) ⇔ *b*=*ByteTag*(0,*Utf8*(*s*))
definition !*b*,*s*,*t*. *Responded*(*b*,*s*,*t*) ⇔ *b*=*ByteTag*(1,*Pair*(*Utf8*(*s*),*Utf8*(*t*)))

Let `format-symb.fs7` be `format.fs7` with these two definitions.

Here is the corresponding implementation.

Symbolic Implementation of Formatting: (`format-symb.fs`**)**

let *request s* = *tag* 0 (*utf8 s*)
let *response s t* = *tag* 1 (*pair* (*utf8 s*) (*utf8 t*))

We typecheck this implementation by running a command of the form

```
> f7 mac-symb.fs7 format-symb.fs7 format-symb.fs
```

This verifies, in particular, that (1) our implementations of *request* and *response* comply with their specifications *Request* and *Response*, and (2) these specifications logical imply the **assert**ed properties exported by our interface for verifying RPC.

To get the details, we may insert the command-line option `-scripts format`, which causes F7 to keep an SMP trace of all its calls to Z3, its underlying prover. We then observe that typechecking *format−symb.fs* involves calling Z3 on nine logical proof obligations: two for checking implementations against specifications (1); four for proving that terms are indeed logical *Bytes*; and three for proving the **assert**ed formulas (2).

5.2 Refinement Types for the RPC Protocol

Using F7, we check that our protocol code (with the *Net* and *Crypto* library interfaces, and the assumed formulas above) is a well-typed implementation of the interface below.

Typed Interface for the RPC Protocol: (`rpc.fs7`)

val *keygen*: *a*:*principal* → *b*:*principal* → *k*:key {*KeyAB*(*k*,*a*,*b*)}
val *corrupt*:
  a:*principal* → *b*:*principal* {*Corrupt*(*a*) ∨ *Corrupt*(*b*)} → *k*:key {*KeyAB*(*k*,*a*,*b*)} → *b*:bytes
val *client*:
  a:*principal* → *b*:*principal* → *k*:key {*KeyAB*(*k*,*a*,*b*)} → *r*:*req*{ *Request*(*a*,*b*,*r*) } → unit
val *server*:
  a:*principal* → *b*:*principal* → *k*:key {*KeyAB*(*k*,*a*,*b*)} →
  (*r*:*req* → *s*:*rsp* {*Request*(*a*,*b*,*r*) ⇒ *Response*(*a*,*b*,*r*,*s*)}) → unit

5.3 Modelling the Symbolic Adversary

As outlined in Section 3.6, we model an active adversary as an arbitrary program well-typed against an adversary interface, I_{adv}. In the symbolic model, we define the interface as follows. Recall that **type** bytespub = *t*:bytes {*Pub*(*t*)}. The function *GENPUB* generates a new key *k*, assumes that $Msg(k,t)$ holds for all *t*, which establishes $Pub(k)$, and returns *k*. The function *princ_corrupt* assumes the event $Corrupt(a)$ for its argument *a*, and returns all the keys associated with *a*; we have $Pub(k)$ for each *k* associated with *a*, because, by definition of *KeyAB* in Section 4.3, the predicate $Msg(k,t)$ holds for all *t*.

Adversary Interface: I_{adv}

val *send*: bytespub → unit
val *recv*: (bytespub → unit) → unit

val *utf8*: string → bytespub
val *iutf8*: bytespub → string

val *tag*: int → bytespub → bytespub
val *itag*: bytespub → int ∗ bytespub

val *pair*: bytespub → bytespub → bytespub
val *ipair*: bytespub → bytespub ∗ bytespub

val *GENPUB*: unit → bytespub
val *MAC*: bytespub → bytespub → bytespub
val *VERIFY*: bytespub → bytespub → bytespub → bool

val *keygen*: *principal* → *principal* → unit
val *client*: *principal* → *principal* → string
val *server*: *principal* → *principal* → (string → string) → unit
val *princ_corrupt*: *principal* → bytespub list

Let a *symbolic adversary Adv* be a program that contains no **assume** or **assert** and that is well-typed against the adversary interface, that is, $I_{adv} \vdash Adv : unit$.

5.4 Symbolic Soundness for the RPC Protocol

We are now ready to formally state our target security theorem for this protocol. We say that an expression is *semantically safe* when every executed assertion logically follows from previously-executed assumptions. For the formal details, see Bhargavan et al. (2010). The proof of the following is by typechecking with F7, as sketched in Section 4.4.

Theorem 2 (Symbolic RPC Safety) *For any symbolic adversary Adv, the module Lib · MAC · Format · RPC · Adv is semantically safe.*

6 Computational Soundness

We now explain how to re-use our refinement-based specification to verify the RPC protocol implementation using a more concrete model of cryptography.

Formally, we need to adapt our language semantics to reason about probabilistic and polynomial-time expressions—we refer the interested reader to Fournet et al. (2011) for a detailed presentation. We also need to develop another, computationally-sound implementation of our *Mac* module given a standard cryptographic assumption, as explained below. On the other hand, computational soundness makes little difference as regards automated protocol verification: we still use the F7 typechecker, and we only need to slightly adapt our F# code and F7 interfaces for the RPC protocol.

In contrast with symbolic models, our reasoning relies on the concrete representation of values of type bytes, as they are used for instance in the system implementation of *Mac* in Section 3.4. We let **type** bytes = *byte array*, where *byte* is the primitive F# type for 8-bit integers. More precisely, we often keep track of the length of byte arrays, so that we can reason about message formats and call cryptographic primitives with correct arguments. Hence, for instance, we let **type** mac = *b*:bytes {*Length*(*b*)= *macsize*} for concrete MAC values, so that we consistently use MACs with exactly *macsize* bytes.

Another difference is that our computational libraries need not share a single definition for all cryptographic terms. In Section 5, the symbolic definition of bytes covers MACs, but also UTF8 encodings and concatenation of bytes; more generally, the symbolic libraries of Bhargavan et al. (2010) jointly define all message and cryptographic processing, using large algebraic data types. In contrast, our computational variants of the modules *Mac* and *Format* are independent, and more generally each cryptographic primitive can be specified and implemented in a separate module (Fournet et al. 2011). On the other hand, computationally-sound interfaces are usually more restrictive than symbolic ones.

We first present our concrete implementation for authenticated message digests, as a first example of byte-level verification, then we discuss computational soundness for MACs using an intermediate, ideal implementation, and we finally give a sample computational security theorem for the RPC code.

6.1 Verifying Binary Message Formats

We write (and verify) another pair of formatting functions, this time using concrete bytes and inserting 2-byte lengths for variable-length fields instead of relying on abstract pairs—these two functions are used for compiling and running the protocol. The implementation is:

F# Concrete Implementation of Formatting: (`format.fs`)

```
module Format
open Lib
let tag0 = [| 0uy |]
let tag1 = [| 1uy |]
let request s = concat tag0 (utf8 s)
let response s t =
  let sb = utf8 s
  concat tag1 (concat (int2bytes (length sb)) (concat sb (utf8 t)))
```

We typecheck this implementation against the interface listed in Section 4.1 supplemented with the declarations below:

```
val tag0: b:bytes { Length(b)=1 }
val tag1: b:bytes { Length(b)=1 ∧ b ≠ tag0 }
definition !v,s. Requested(v,s) ⇔ v = tag0 | Utf8(s)
definition !v,s,t. Responded(v,s,t) ⇔ v = tag1 | Int2Bytes(Length(Utf8(s ))) | Utf8(s) | Utf8(t)
```

The refinements on tags explicitly state that they are distinct one-byte arrays; the logical definitions give a complete specification of our digest formats. The automated verification relies on precise type annotations in our library to keep track of the lengths of concatenated byte arrays. For instance, the second **assert**ed property of the interface logically follows by inlining the definition of *Requested* on both sides of $v = v'$, and then applying our axiom concerning equality of concatenations when the length of the first byte arrays are the same.

6.2 A Concrete Refined Interface for MACs

We first give more precise types to the concrete MAC implementation of Section 3.4. These types reflect basic assumptions on the underlying cryptographic algorithms, which are outside the scope of this code verification effort; they are used in the proof but they are not sufficient to ensure message authentication. We use the interface:

```
type mac = b:bytes{Length(b) = macsize}
predicate val GENerated: key → bool
predicate val MACed: key * text * mac → bool
val GEN: unit → k:key {GENerated(k)}
val MAC: k:key → t:text → m:mac {MACed(k,t,m)}
val VERIFY: k:key → t:text → m:mac →
  v:bool{ GENerated(k) ∧ MACed(k,t,m) ⇒ v = true }
val LEAK: k:key → b:bytes{Length(b) = keysize}
assume !k,t,m0,m1.
  GENerated(k) ∧ MACed(k,t,m0) ∧ MACed(k,t,m1) ⇒ m0 = m1
```

In this interface, written I^C in the following, two auxiliary predicates *GENerated* and *MACed* keep track of the results returned by *GEN* and *MAC*. Relying on these predicates, the post-condition of *VERIFY* states a functional correctness property: verification of a MAC computed using matching keys and texts always succeeds; and the final assumption states that *MAC* is a deterministic function.

These properties are easily met by any implementation that just recomputes the MAC for verification, as our concrete implementation does: to match I^C, it suffices to add events at the end of *GEN* and *MAC* to record their result, and to rely on the functionality of *MAC* to establish the post-condition of *VERIFY*.

6.3 Resistance to Existential Chosen-Message Forgery Attacks (CMA)

We now describe our computational notion of security for MACs, expressed as a game between the cryptographic functionality and a probabilistic polynomial-time adversary. We adopt a standard notion, named *resistance against existential Chosen-Message forgery Attacks* (CMA) and first proposed by Goldwasser et al. (1988) for signatures; we follow the presentation of Bellare et al. (1996; 2005).

As usual with concrete cryptography, the MAC key is a finite bitstring, so we cannot entirely exclude that a lucky adversary may just guess the key, or some valid MACs, using for instance random sampling. Instead, our notion of security is relative to a security parameter, which informally controls the length of the key and other cryptographic values, and we bound the probability that an adversary succesfully forges a MAC: we say that a probabilistic, polynomial-time program is *asymptotically safe* when the probability that any logical assertion fails at runtime is *negligible*, that is, the probability is a function of the security parameter that converges to zero faster than the inverse of any polynomial.

In the game that defines CMA, an adversary attempts to forge a valid MAC; to this end, it can adaptively call two *oracles* for computing and verifying MACs on a single, hidden key; the adversary wins if it can produce a text and a MAC such that verification succeeds but the text has not been passed to the MAC oracle. We set up this game using an F# module, written *CMA* and defined as follows:

This code generates a key, implements the two oracles as functions using that key, and maintains a log of all texts passed as arguments to the mac function. We intend to run this code with an adversary given access to mac and *verify*, but not *k* or *log*. The verification oracle also tests whether any verified pair wins the game: the assert claims that this does not happen: either *v* is false, or *t* is in the log. (In the asserted formula, || is logical disjunction.)

Definition 1. *A Mac implementation C is CMA-secure when, for all p.p.t. expressions A with no* ***assume*** *or* ***assert*** *such that mac: text* $\rightarrow$ *mac, verify: text* $\rightarrow$ *mac* $\rightarrow$ *bool* $\vdash$ *A: unit, the expression* $C \cdot CMA \cdot A$ *is asymptotically safe.*

We recall the two main security theorems of Fournet et al. (2011) for MACs, which establish asymptotic safety for programs that are well-typed against the *ideal* interface

of a CMA-secure MAC, detailed in Section 4.2 and written I_{MAC} below. Crucially, although keys are just bytes, their type in I_{MAC} is abstract, so that protocols cannot directly generate or access their values.

Theorem 3 (Asymptotic Safety for MAC) *Let C be a p.p.t. implementation of the refined concrete interface: $\vdash C \rightsquigarrow I^C$. Let A be a p.p.t. expression such that $I \vdash A : unit$. If C is CMA-secure, then $C \cdot A$ is asymptotically safe.*

We will apply this theorem to show that our sample RPC code is secure against all active probabilistic polynomial-time adversaries, but first we explain how it is established by programming and typechecking.

6.4 An Ideal Functionality for MACs

Our main tool is an intermediate implementation of *Mac*, written F and defined below, such that $C \cdot F$ is well-typed against the ideal interface I_{MAC}. (We call F an *ideal functionality* for MACs, and call $C \cdot F$ an ideal MAC implementation; in terms of universal composability, C emulates F, with C as a simulator.) A second security theorem states that the concrete and ideal implementations are *indistinguishable* ($\approx_\varepsilon$), that is, the probability that a probabilistic polynomial adversary guesses which of the two implementations it is running with minus $\frac{1}{2}$ (the probability of guessing at random) is negligible.

Theorem 4 (Ideal Functionality for MAC) *Let C be a p.p.t. CMA-secure implementation of the concrete-refined interface I^C. Let A be p.p.t. and such that $I \vdash A : bool$. We have $C \cdot A \approx_\varepsilon C \cdot F \cdot A$.*

The proof relies on a standard code-based cryptographic argument to relate the code of F to the similar, but simpler code that defines *CMA* security (see Fournet et al. 2011).

Intuitively, the ideal implementation is thus both perfectly secure and indistinguishable from the concrete one. This is achieved by intercepting all calls to the concrete implementation and correcting the result of concrete verification from **true** to **false** when the message has not been MACed and the key has not been leaked—our CMA assumption implies that such corrections occur with negligible probability.

Our ideal functionality re-defines the type for keys, now represented as integer indexes, and re-defines the functions *GEN*, *MAC*, *VERIFY*, and *LEAK* using those provided by any implementation of the refined concrete interface I^C. It maintains a global, private association table *ks* that maps each key index to some internal state, consisting of a concrete MAC key (*kv*), a list of previously-MACed texts (*log*), and a boolean flag (*leaked*) indicating whether the key has been leaked or not. In the code below, qualified names such as *Mac.GEN* refer to entries from the concrete interface I^C, while names such as *GEN* refer to their re-definition.

Ideal Functionality for MACs: (`macf.fs`)

```
let ks = ref [] (* state for all keys generated so far *)
let GEN () =
  let k = list_length !ks
  let kv = Mac.GEN() in
  ks := (k,(kv,empty_log k,false_flag k))::!ks; k
let MAC k t =
  let (kv,log,leaked) = assoc k !ks in
  log := t:: !log; Mac.MAC kv t
let VERIFY k t m =
  let (kv,log,leaked) = assoc k !ks in
  Mac.VERIFY kv t m && (mem k t !log || !leaked)
let LEAK k =
  let (kv,log,leaked) = assoc k !ks in
  leaked := true; Mac.LEAK kv
```

In this code, the auxiliary functions *list_length*, *assoc*, and *mem* are the standard F# functions *List.length*, *List.assoc*, and *List.mem* with more precise refinement types, and the functions *empty_log* and *false_flag* allocate mutable references that initially contain an empty list and **false**, respectively. The four main functions are as follows:

- *GEN* allocates a key with index k (the number of keys allocated so far), generates a concrete key, and records its initial state, with an empty log and a flag set to **false**. When called with a key, the three other functions first perform a table lookup to retrieve its state (*kv*,*log*,*leaked*).
- *MAC* computes the MAC using the concrete *kv* and adds the MACed text to the log.
- *VERIFY* performs the concrete MAC verification, then it corrects its result from **true** to **false** when (i) the concrete verification succeeds; but (ii) the text has not been MACed, as recorded in *log*; and (iii) the key has not been leaked, as recorded by *leaked*.
- *LEAK* returns the concrete key bytes, and sets the *leaked* flag to **true**.

The state entries in the table *ks* are typed using the following auxiliary declarations:

```
type (;k:key) msg = t:text{ Msg(k,t) }
type (;k:key) leak = b:bool{ b=true ⇒ !t. Msg(k,t) }
type (;k:key) state = Mac.key * (;k) msg list ref * (;k) leak ref
val ks: (k:key * (;k) state) list ref
```

These types enforce runtime invariants: all texts t kept in the log for key k are such that $Msg(k,t)$, and we have $!t.Msg(k,t)$ whenever the flag is set to **true**. (In F7 syntax, (;*k*:key) *msg* declares a type whose refinements depend on the value k.)

We automatically typecheck $I^C \vdash F \rightsquigarrow I$; intuitively, this holds because the result of the verification function is always consistent with the log and the flag, whose refinement types record enough facts $Msg(k,t)$ to establish the post-condition of *VERIFY*. We have also supposed that $\vdash C \rightsquigarrow I^C$, so the module $C \cdot F$ obtained by composing the concrete MAC implementation C with F is a well-typed implementation of the ideal interface I_{MAC}, as F corrects any unsafe verification of C, but it is unrealistic, as F relies on global shared state.

We conclude this development with an informal proof of Theorem 3 (Asymptotic Safety for MAC). For a given A such that $I_{MAC} \vdash A$, we obtain $\vdash C \cdot F \cdot A$ by composing our typing judgments, so $C \cdot F \cdot A$ is perfectly safe, by type safety. Moreover, by Theorem 4 (Ideal Functionality for MAC), $C \cdot F \cdot A$ and $C \cdot A$ are indistinguishable, so $C \cdot A$ is also safe except with a negligible probability.

6.5 Computational Soundness for the RPC Protocol

We give on the next page a variant of our code of the RPC protocol. This code differs from that presented in Section 3 only in its use of the network: in the computational model of computation, we cannot rely on parallel threads to send and receive messages; instead, we let the adversary schedule the two stages of the client and the server. On the other hand, the code for generating and verifying messages is unchanged.

For the second stage of the client, *client2*, the type interface is as follows

val *client2*:
a:*principal* $\to$ b:*principal* $\to$ k:key $\{KeyAB(k,a,b)\} \to$
r:*req*$\{$ *Request*(a,b,r) $\} \to msg \to$
(s:*rsp*$\{$ *Response*$(a,b,r,s) \vee$ *Corrupt*$(a) \vee$ *Corrupt*(b) $\}$) *option*

Let *Net* range over modules implementing the network and the scheduler, informally controlled by the adversary, with the following interface for the protocol:

$$I_{Net} \stackrel{\triangle}{=} \mathit{send}\text{: bytes} \to \text{unit}, \mathit{recv}\text{: (bytes} \to \text{unit)} \to \text{unit}$$

In addition, *Net* may export an arbitrary interface I'_{NET} for the adversary. Let C_{RPC} be the sample protocol code above, after inlining its other (trusted) library functions, with the following interface for the adversary: $I_{\mathsf{RPC}} \stackrel{\triangle}{=} \mathit{client}\text{: string} \to \text{unit}, \mathit{server}\text{: unit} \to \text{unit}$

We arrive at the following computational safety theorem:

Theorem 5 (Computational RPC Safety) *Let C be a CMA-secure p.p.t. module with no assert such that $\vdash C \rightsquigarrow I^C$. Let Net be a module with no assert such that $\vdash Net \rightsquigarrow I_{\mathsf{NET}}$, $\vdash Net \rightsquigarrow I'_{\mathsf{NET}}$, and $Net \cdot C_{\mathsf{RPC}}$ is p.p.t. Let A be a polynomial expression with no assert such that $I'_{\mathsf{NET}}, I_{\mathsf{RPC}} \vdash A : unit$.*

The expression $C \cdot Net \cdot C_{\mathsf{RPC}} \cdot A$ is asymptotically safe.

(Technically, we state our p.p.t. assumption on *Net* composed with C_{RPC} because *Net* has a second-order interface.) In case I'_{NET} consists of first-order functions on bytes, and for each concrete definition of *Net*, we obtain as a special case a computational safety theorem for an adversary A with traditional oracle access to the network and the protocol. Typability of A is then irrelevant, since A exchanges only raw bytes with the rest of the system and may implement any polynomial computation on those bitstrings.

Although we now rely on computational assumptions, our protocol code still typechecks essentially for the same reasons as in Section 5. Compared with our symbolic theorem, the computational theorem involves a more complex semantics, but a simpler adversary interface: the computational adversary need not be given access to the cryptographic library or formatting functions such as *concat*, since it may instead include its own concrete implementation of MAC and formatting algorithms.

F# Implementation for the Authenticated RPC Protocol: (`rpc.fs`)

```
module RPC
open Pi
open Lib
open Mac
open Format

type principal = string
type req = string
type rsp = string
type msg = bytes
type fact = KeyAB of key * principal * principal

let keygen (a:principal) (b:principal) =
  let k = GEN() in assume(KeyAB(k,a,b)); k

let corrupt (a:principal) (b:principal) k = LEAK k

let client (a:principal) (b:principal) k s =
  let msg1 = concat (utf8 s) (MAC k (request s))
  send msg1
let client2 (a:principal) (b:principal) k s msg2 =
  let (v,m') = split macsize msg2 in
  let t = iutf8 v in
  if VERIFY k (response s t) m' then Some(t) else None

let server (a:principal) (b:principal) k service =
  recv (fun (msg1:bytes) →
    let (v,m) = split macsize msg1 in
    let s = iutf8 v in
    if VERIFY k (request s) m then
      let t = service s
      let msg2 = concat (utf8 t) (MAC k (response s t))
      send msg2)
```

7 Further Reading

A companion tutorial (Gordon and Fournet 2010) focuses on the type theory underlying F7, and includes pointers for further reading on operational semantics, type systems, process calculi, and refinement types. That tutorial also discusses some initial applications of F7, including synthesis of security protocols for multiparty sessions (Bhargavan et al. 2009b), security of multi-tier web programming (Baltopoulos and Gordon 2009), auditability by typing (Guts et al. 2009), and the use of the refined state monad to verify imperative F# code (Borgström et al. 2011), a line of work resulting in the use of F7 to verify that SQL transactions maintain database integrity (Baltopoulos et al. 2011). Recent applications of F7 to cryptographic software include verifications for F# reference

implementations of the XML-based CardSpace protocol (Bhargavan et al. 2008a, 2010) and of the DKM key management library (Acar et al. 2011).

The present tutorial introduces F7 as a tool for verifying cryptographic software in both the symbolic and computational models. We have illustrated an approach to computational verifications based on the work of Fournet et al. (2011); Backes et al. (2010) develop an alternative approach, in which they obtain computationally sound verifications of F7 programs by appeal to the general CoSP framework. F7 does not support union or intersection types; Backes et al. (2011) extend the type theory of F7 with these features and show their utility for dealing with public-key cryptography and zero-knowledge proofs. In other recent work, Swamy et al. (2011) present the design and implementation of F*, which integrates the refinement types of F7 into the Fine language (Swamy et al. 2010), and hence supports cryptographic software. In earlier work on protocols in F#, Bhargavan et al. (2008b) develop FS2PV, which analyzes F# software by compilation to ProVerif (Abadi and Blanchet 2005).

The pioneers of cryptographic software verification are Goubault-Larrecq and Parrennes (2005), whose tool Csur analyzes secrecy properties of C code via a custom abstract interpretation. Another verification tool for C is ASPIER (Chaki and Datta 2009), which relies on various abstractions and on software model-checking techniques. Dupressoir et al. (2011) recast the F7 approach to symbolic verification of protocol code in the setting of the general-purpose C verifier VCC (Cohen et al. 2009) so as to verify protocol code in C; their work is the first to prove memory safety and security properties simultaneously for C. (F7 need only prove security properties, as the underlying F# type system proves memory safety directly.) Other approaches to C cryptographic protocols include: automatic generation of protocol monitors, to detect if production protocols deviate from their specification, from abstract protocol descriptions Pironti and Jürjens (2010), and the use of symbolic execution (Corin and Manzano 2011; Aizatulin et al. 2011) to extract potentially verifiable symbolic representations of cryptographic computations in C.

In summary, research on verifying cryptographic implementations is an active area. As this tutorial illustrates, F7 shows the viability of refinement types for verifying security properties of F# code in both the symbolic and computational models. It remains a research challenge to scale up verification to large, existing codebases, and to languages more commonly used for cryptographic software, including C, Java, C#, and even JavaScript.

References

M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *JACM*, 52(1):102–146, 2005.

M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

T. Acar, C. Fournet, and D. Shumow. DKM: Design and verification of a crypto-agile distributed key manager. Available at `http://research.microsoft.com/en-us/um/people/fournet/dkm/`, 2011.

M. Aizatulin, A. D. Gordon, and J. Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In submission, 2011.

M. Albrecht, K. Paterson, and G. Watson. Plaintext recovery attacks against SSH. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 16 –26, may 2009.

M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *ACM CCS*, pages 387–398, 2010.

M. Backes, C. Hriţcu, and M. Maffei. Union and intersection types for secure protocol implementations. In *TOSCA*, 2011.

I. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2009)*, pages 27–38, 2009.

I. Baltopoulos, J. Borgström, and A. D. Gordon. Maintaining database integrity with refinement types. In *ECOOP 2011*, 2011.

M. Bellare and P. Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, 2005.

M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.

K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the Information Card federated identity-management protocol. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 123–135. ACM Press, 2008a.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31:5:1–5:61, December 2008b.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140. IEEE Computer Society, 2009a.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 124–140. IEEE Computer Society, July 2009b.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456, 2010.

D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In H. Krawczyk, editor, *Advances in Cryptology - CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. *Journal of Functional Programming*, 21(2):159–207, 2011.

I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad. Breaking and fixing public-key Kerberos. *Information and Computation*, 206(2-4):402–424, 2008.

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE Computer Security Foundations Symposium*, pages 172–185, 2009.

E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.

R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *ESSoS*, volume 6542 of *LNCS*, pages 58–72, 2011.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

F. Dupressoir, A. Gordon, J. Jürjens, and D. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *24th IEEE Computer Security Foundations Symposium*, 2011. To appear.

D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), 2001. RFC 3174.

C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. Technical report, sample code, and formal proofs available from `http://research.microsoft.com/~fournet/comp-f7/`, 2011.

S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17(2):281–308, 1988.

A. D. Gordon and C. Fournet. Principles and applications of refinement types. In J. Esparza, B. Spanfelner, and O. Grumberg, editors, *Logics and Languages for Reliability and Security*. IOS Press, 2010. Available as Microsoft Research Technical Report MSR–TR–2009–147.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, volume 3385 of *LNCS*, pages 363–379, 2005.

N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability by typing. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 168–183. Springer, 2009.

H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, 1997. RFC 2104.

G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

A. Pironti and J. Jürjens. Formally-based black-box monitoring of security protocols. In *ESSoS*, pages 79–95, 2010.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP*, pages 529–549, 2010.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP 2011)*, 2011. To appear.

S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In L. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–545. Springer, 2002.